Multi-Stage Programs are Generalized Arrows

This paper is obsolete and has been superceded by
Multi-Level Programs are Generalized Arrows
available here:
http://arxiv.org/pdf/1007.2885

Adam Megacz
UC Berkeley
megacz@berkeley.edu

Abstract

The lambda calculus, subject to typing restrictions, provides a syntax for the internal language of cartesian closed categories. This paper establishes a parallel result: staging annotations [**?**], subject to named level restrictions, provide a syntax for the internal language of Freyd categories, which are known to be in bijective correspondence with `Arrows`. The connection is made by interpreting multi-stage type systems as indexed functors from polynomial categories to their reindexings (Definitions 16 and 17).

This result applies only to multi-stage languages which are (1) homogeneous, (2) allow cross-stage persistence and (3) place no restrictions on the use of structural rules in typing derivations. Removing these restrictions and repeating the construction yields *generalized arrows*, of which `Arrows` are a particular case. A translation from well-typed multi-stage programs to single-stage `GArrow` terms is provided. The translation is defined by induction on the structure of the proof that the multi-stage program is well-typed, relying on information encoded in the proof's use of structural rules (weakening, contraction, exchange, and context associativity).

Metalanguage designers can now factor out the syntactic machinery of metaprogramming by providing a single translation from staging syntax into expressions of generalized arrow type. Object language providers need only implement the functions of the generalized arrow type class in point-free style. Object language users may write metaprograms over these object languages in a point-ful style, using the same binding, scoping, abstraction, and application mechanisms in both the object language and metalanguage.

This paper's principal contributions are the `GArrow` definition of Figures 2 and 3, the translation in Figure 5 and the category-theoretic semantics of Definition 16. An accompanying Coq proof formalizes the type system, translation procedure, and key theorems.

1. Introduction

Metaprogramming, the practice of writing programs which construct and manipulate other programs, has a long history in the computing literature. However, prior to [**?**] little of it dealt with metaprogramming in a statically typed setting where one wants to ensure not only that "well typed programs do not go wrong," but also that well typed metaprograms *do not produce ill-typed object programs*.

One of the most popular applications of statically typed metaprogramming has been the use of monads to account for different *notions of computation* [**?**] as the impure programs manipulated by pure functions in a category equipped with a Kleisli triple. The use of monads in functional programming was later generalized to `Arrows` by Hughes, who writes "every time we sequence two monadic computations, we have an opportunity to run arbitrary code in between them. [**?**]" `Arrows` curtail this freedom, permitting the inclusion of static information. In practice, this has made `Arrows` a popular framework for metaprogramming, particularly when one is allowed to do things with object programs other than run them.

Because adding a new object language involves nothing more than implementing the functions required by the `Arrow` type class, this approach to embedding makes it quite easy to *provide* new object languages. Although all embedded languages share a common syntax [**?**], this syntax is profoundly different from that of the metalanguage, which can make it difficult to *use* object languages.

By contrast, staging annotations [**?**] embed an object language within the metalanguage using the same binding, scoping, abstraction, and application mechanisms as the metalanguage. However, the type system of the metalanguage must reflect the type system of the object language, so adding a new object language is quite difficult and generally requires making modifications to the metalanguage compiler.

This paper will use, as a running example, the `pow` function which has become ubiquitous in the metaprogramming literature. Here is the `pow` program written using `Arrow` notation [**?**]:

```
Class Arrow
           ((~>):Set->Set->Set) :=









 arr    : (a->b) -> (a~>b)

 (>>>)  : a~>b -> b~>c -> a~>c
 first  : a~>b -> (a*c)~>(b*c)

 (~~)   : a~>b -> a~>b -> Prop

 pf1    : Equivalence (a~~b)
 pf2    : Morphism (a~~b ==> b~~c ==> a~~c) (>>>)
 pf3    : Morphism (a~~b ==> (a*c)~~(b*c)) first
```

Figure 1. Definition for the `Arrow` class. See also Remark 1.

```
Class GArrow ((**):Set->Set->Set)
             ((~>):Set->Set->Set) :=

 id     :          a ~> a
 assoc  : (a**b)**c ~> a**(b**c)
 cossa  : a**(b**c) ~> (a**b)**c
 copy   :          a ~> a**a
 drop   :       a**b ~> a
 swap   :       a**b ~> b**a

 (>>>)  : a~>b -> b~>c -> a~>c
 first  : a~>b -> (a**c)~>(b**c)

 (~~)   : a~>b -> a~>b -> Prop

 pf1    : Equivalence (a~~b)
 pf2    : Morphism (a~~b ==> b~~c ==> a~~c) (>>>)
 pf3    : Morphism (a~~b ==> (a**c)~~(b**c)) first
```

Figure 2. Definition for the `GArrow` class. See also Remark 1.

```
pow n =
  if n==0
  then cst 1
  else proc x ->
       do pow'    <- (pow (n-1)) -< x
          result  <- (*)         -< (x, pow')
          returnA -< result
```

Here is an equivalent program written using staging annotations:

```
pow n x =
  if n==0
  then <[ 1 ]>
  else <[ ~x * ~(pow (n-1) x) ]>
```

Section 2 reviews `Arrows` and introduces generalized arrows. Section 3 presents a grammar and type system for a simplified MetaML-style [**?**] multi-stage programming language. Section 4 provides a translation procedure which produces generalized arrow values from the typing derivations of well-typed multi-stage programs. Section 5 walks through a few example programs, and Section 6 formalizes the category-theoretic underpinnings of staging annotations.

2. `Arrows`

From a programmer's perspective, an `Arrow` is a type belonging to the Coq type class [**?**] shown in Figure 1. Briefly, the members of the class are type operators `(~>)` which take two arguments, supplied along with a function `arr` which lifts arbitrary functions into `Arrows`, a function `(>>>)` which composes `Arrows`, and a function `first` which lifts an `Arrow` on a given type to an `Arrow` on tuples with that type as the first coordinate and the identity operation on the second coordinate. The last four declarations define an equivalence relation `(~~)` and require that `(>>>)` and `first` preserve it.

Remark 1 To improve readability, the following elements of Coq syntax have been elided from the printed version of this paper: semicolons, curly braces, `Notation` clauses, `Implicit Argument` clauses, explicit instantiation of implicit arguments, and polymorphic type quantifiers (specifically, `forall` occurring immediately after a colon). The complete Coq code, which includes the elided text, is available online[1]

2.1 Generalized `Arrows` (`GArrows`)

The Coq declaration for the `GArrow` class is shown in Figure 2; the laws for `GArrows` can be found in Figure 3 using mathematical notation, and in Figure 15 using Coq notation. Proofs of these propositions appear as obligations for any code attempting to create an instance of the `GArrow` class, providing machine-checked assurance that the laws are satisfied.

Comparing the two declarations, one can see that `GArrows` generalize `Arrows` in two ways:

1. The `arr` constructor is omitted, and part of its functionality is restored via `id`, `assoc`, `cossa`, `drop`, `copy`, and `swap`.
2. The methods of the `Arrow` class are specified in terms of tuple types, which are assumed to be full cartesian products. `GArrows` relax this restriction, assuming only that the tupling operator is a monoid.

Parameterizing `GArrow` over an arbitrary `(**):Set->Set->Set` operator rather than requiring the use of the cartesian product allows for more generality: while there is a straightforward function of type $(\forall\alpha)\alpha\to(\alpha,\alpha)$, there is no total function of type $(\forall$`(**):Set->Set->Set`$)(\forall\alpha)\alpha\to(\alpha$`**`$\alpha)$. The weaker construct makes it possible to deny users the ability to form such functions where they are inappropriate. In particular, it prevents properties of the cartesian product from imposing unwanted properties upon object language contexts, as will be shown in Definition 16 and utilized in Section 5.2.

Remark 2 The following `Arrow` laws from [**?** , Figure 1] have been omitted from `GArrow` because they serve only to regulate `arr`:

$$\texttt{arr}(g \circ f) = \texttt{arr}\ f \ggg \texttt{arr}\ g \qquad (10)$$

$$\texttt{first}(\texttt{arr}\ f) = \texttt{arr}(f \times \texttt{id}) \qquad (11)$$

$$\texttt{first}\ f \ggg \texttt{arr}\ (\texttt{id} \times g) = \texttt{arr}\ (\texttt{id} \times g) \ggg \texttt{first}\ f \qquad (12)$$

However, (11) above does serve the same purpose as law (7) of Figure 3.

[1] `http://www.cs.berkeley.edu/~megacz/garrows/GArrow.v`

$$\mathtt{id} \ggg f = f \tag{1}$$

$$f \ggg \mathtt{id} = f \tag{2}$$

$$(f \ggg g) \ggg h = f \ggg (g \ggg h) \tag{3}$$

$$\mathtt{first}\ (f \ggg g) = (\mathtt{first}\ \ f) \ggg (\mathtt{first}\ \ g) \tag{4}$$

$$\mathtt{first}\ (\mathtt{first}\ f) \ggg \mathtt{assoc} = \mathtt{assoc} \ggg \mathtt{first}\ f \tag{5}$$

$$\mathtt{cossa} = \mathtt{swap} \ggg \mathtt{assoc} \ggg \mathtt{swap} \tag{6}$$

$$\mathtt{first}\ f \ggg \mathtt{drop} = \mathtt{drop} \ggg f \tag{7}$$

$$\mathtt{swap} \ggg \mathtt{swap} = \mathtt{id} \tag{8}$$

$$\mathtt{copy} \ggg \mathtt{swap} = \mathtt{copy} \tag{9}$$

Figure 3. Generalized Arrow laws. The first five laws are taken from [**?** , Figure 1]. The sixth law defines `cossa` in terms of `swap`; this makes it a redundant operation (much like `***` for `Arrows`), though Section 4.6 investigates variants which eschew `swap`, making `cossa` no longer redundant. The seventh law expresses the fact that `first` should not have side effects. The last two laws establish some straightforward properties of `swap` and `copy`. A Coq rendition of these laws can be found in Figure 15.

Theorem 1 Every `Arrow` is a `GArrow prod`, where `prod` is the cartesian product.

Proof. `Instance Arrows_are_GArrows` in `GArrow.v` □

3. Staging Annotations

3.1 Natural Deduction

This section briefly reviews the structural rules for natural deduction. Δ will denote derivations, Σ will denote propositions and Γ will denote contexts, where a context consists either of a single proposition or a pair of subcontexts:

$$\Gamma ::= \Sigma \mid \Gamma, \Gamma$$

Therefore contexts can be viewed as binary trees.

Remark 3 Although logically quite conventional – the $(\cdot,\cdot)$ construct is exactly logical conjunction – this choice is proof-theoretically nonstandard; contexts are usually handled as lists. However, the translation given in Section 4 is only valid for proof derivations which are completely explicit about every structural rule invocation. The positions of these invocations in the proof derivation carry information which is used by the translation.

By representing contexts with binary trees rather than lists one can avoid introducing rules which *implicitly* rearrange the context. One example of such a rule is one which uses ellipsis to abbreviate a sequence of propositions:

$$\Gamma, \ldots, x : \tau \vdash \Sigma$$

Another example is a rule which tacitly assumes that lists of hypotheticals are identified up to associativity:

$$\Gamma_1, x : \tau, \Gamma_2 \vdash \Sigma$$

The first six rules of Figure 5 are the structural rules. They are allow all other rules to be in a form where any necessary assumptions appear as the leftmost child of the context.

$\Sigma ::= \top \mid e : \tau^{\vec{\eta}} \mid \mathsf{firstClass}(\tau, \vec{\eta})$	$e ::= x \mid \lambda x.e \mid e[\vec{e}] \mid \langle\!|e|\!\rangle \mid \sim\! e$
$\Gamma ::= \Sigma \mid \Gamma, \Gamma$	$\vec{e} ::= \cdot \mid e, \vec{e}$
$\eta ::=$ level name	$x ::=$ expression variable
$\vec{\eta} ::= \cdot \mid \eta, \vec{\eta}$	$\tau ::= \tau \to \tau \mid \langle\!|\tau^{\eta}|\!\rangle$

Figure 4. Grammar for a simple multi-stage language.

Lemma 1 (Permutation of Contexts) If there is a proof terminating in the judgement

$$\frac{\vdots}{\Gamma_1 \vdash \Sigma_1}$$

and some proposition Σ_2 appears as a leaf of Γ_1, then there is a proof terminating in the judgement

$$\frac{\vdots}{\Sigma_2, \Gamma_2 \vdash \Sigma_1}$$

where the leaves of Σ_2, Γ_2 are a permutation of the leaves of Γ_1. Furthermore, there is an algorithm for transforming the first proof tree into the second.

Proof. in `permutation_of_contexts` in `GArrow.v` □

3.2 Typing Rules for Staging Annotations

The grammar for a simple multi-stage language can be found in Figure 4; the corresponding typing rules are in Figure 5.

Remark 4 Special attention should be paid to the superscripts used to denote levels; a proposition $e : \tau^{\vec{\eta}}$ attributes a type τ to an expression e at a named level $\vec{\eta}$; the named level $\vec{\eta}$ is part of the proposition, not the type. Named levels do not appear as part of types except the code type $\langle\!|\tau^{\eta}|\!\rangle$, which include exactly one level as part of the type; this level is written *inside* the code-brackets. The mnemonic justification for this choice of syntax can be seen in the typing rules for Brak and Esc.

The first nonstructural rule, FC, distinguishes types inhabited by *first class* values – those that can be arguments or return values of functions. Because $\mathsf{firstClass}(\tau{\to}\tau, \vec{\eta})$ is underivable without additional rules, the type system as shown will prohibit first-class functions. However, this restriction can easily be lifted by simply adding another typing rule:

$$\frac{\mathsf{firstClass}(\tau_1, \vec{\eta}) \quad \mathsf{firstClass}(\tau_2, \vec{\eta})}{\mathsf{firstClass}(\tau_1{\to}\tau_2, \vec{\eta})}$$

The next two rules are the variable (Var) and abstraction (Lam) rules. Note that the Var rule is applicable only when the context contains *exactly* the assumption needed and no others. Any extraneous context elements must be explicitly removed using Weak; this will be significant in Section 4.6 which explores the possibility of removing the Weak rule. The Lam rule is standard, save for the additional $\mathsf{firstClass}(\tau_x, \vec{\eta})$ hypothesis; this ensures that abstractions over non-first-class values may not be formed.

The App_0 and App_{n+1} provide for n-ary function application via the $e[\vec{e}]$ production in the grammar. After typechecking is complete, this n-ary application can be syntactically expanded into n instances of (curried) 1-ary application – for example, $e[e_1, e_2, e_3, \cdot]$ becomes $(((ee_1)e_2)e_3)$. However, by having syntactic indication of the application arity available *at typechecking time* the type

RULE	SYNTAX	SEMANTICS
Assoc	$\dfrac{\Gamma_1,(\Gamma_2,\Gamma_3)\vdash\Sigma}{(\Gamma_1,\Gamma_2),\Gamma_3\vdash\Sigma}$	$=\ \Delta$ $=$ `assoc` >>> Δ
Cossa	$\dfrac{(\Gamma_1,\Gamma_2),\Gamma_3\vdash\Sigma}{\Gamma_1,(\Gamma_2,\Gamma_3)\vdash\Sigma}$	$=\ \Delta$ $=$ `cossa` >>> Δ
Exch	$\dfrac{\Gamma_1,\Gamma_2\vdash\Sigma}{\Gamma_2,\Gamma_1\vdash\Sigma}$	$=\ \Delta$ $=$ `swap` >>> Δ
Exch2	$\dfrac{(\Gamma_1,\Gamma_2),\Gamma_3\vdash\Sigma}{(\Gamma_2,\Gamma_1),\Gamma_3\vdash\Sigma}$	$=\ \Delta$ $=$ (`first swap`) >>> Δ
Weak	$\dfrac{\Gamma_1\vdash\Sigma}{\Gamma_1,\Gamma_2\vdash\Sigma}$	$=\ \Delta$ $=$ `drop` >>> Δ
Cont	$\dfrac{\Gamma_1,\Gamma_1\vdash\Sigma}{\Gamma_1\vdash\Sigma}$	$=\ \Delta$ $=$ `copy` >>> Δ
FC	$\dfrac{\mathsf{firstClass}(\tau,(\eta,\vec{\eta}))}{\mathsf{firstClass}(\langle\!|\tau^{\eta}|\!\rangle,\vec{\eta})}$	
Var	$\dfrac{}{x:\tau^{\vec{\eta}}\vdash x:\tau^{\vec{\eta}}}$	$=$ `id`
Lam	$\dfrac{\mathsf{firstClass}(\tau_x,\vec{\eta}) \qquad x:\tau_x^{\vec{\eta}},\Gamma\vdash e:\tau^{\vec{\eta}}}{\Gamma\vdash\lambda x.e:(\tau_x\rightarrow\tau)^{\vec{\eta}}}$	$=\ \Delta$ $=\ \Delta$
App_0	$\dfrac{\mathsf{firstClass}(\tau,\vec{\eta}) \qquad \Gamma\vdash e:\tau^{\vec{\eta}}}{\Gamma\vdash e[\cdot]:\tau^{\vec{\eta}}}$	$=\ \Delta$ $=\ \Delta$
App_{n+1}	$\dfrac{\mathsf{firstClass}(\tau_0,\vec{\eta}) \qquad \Gamma_x\vdash e_x:(\tau_0\rightarrow\tau_x)^{\vec{\eta}} \qquad \Gamma_0\vdash e_0:\tau_0^{\vec{\eta}} \qquad x:\tau_x^{\vec{\eta}},\Gamma_e\vdash x[\vec{e}]:\tau^{\vec{\eta}}}{\Gamma_x,(\Gamma_0,\Gamma_e)\vdash e_x[e_0,\vec{e}]:\tau^{\vec{\eta}}}$	$=\ \Delta_x$ $=\ \Delta_0$ $=\ \Delta_1$ $=$ `first` Δ_0 >>> `second` Δ_1 >>> Δ_x
Brak	$\dfrac{\Gamma\vdash e:\tau^{\eta,\vec{\eta}}}{\Gamma\vdash\langle\!|e|\!\rangle:\langle\!|\tau^{\eta}|\!\rangle^{\vec{\eta}}}$	$=\ \Delta$ $=\ \Delta$
Esc	$\dfrac{\Gamma\vdash e:\langle\!|\tau^{\eta}|\!\rangle^{\vec{\eta}}}{\Gamma\vdash{\sim}e:\tau^{\eta,\vec{\eta}}}$	$=\ \Delta$ $=\ \Delta$

Figure 5. Typing rules for a simple multi-stage language, along with a translation into generalized arrows. The rules and translations are rendered in the rule/syntax/semantics table style of [**?** , Tables 3,5,9]. Note that contexts are represented as a binary tree rather than a list. An explanation of the rules can be found in Section 3.2.

```
Definition pow : E V :=
 letrec pow := \\ n => \\ x =>
      If   (Eeq V) [ ‘n ; (Ezero V) ]
      Then <[Eone V]>
      Else <[(Emult V)[ ~~‘x ;
              (~~ ((‘pow) [ (Eminus V)[ ‘n ;
                 (Eone V)] ; ‘x ])) ] ]>
  in ‘pow.

Eval compute in (translate (pow_hastype _ n)).
letrec x := \\ x0 => \\ x1 =>
   If (first (‘x0)
       >>> second ((first ga_true >>> second id)
                   >>> id))
       >>> ga_true
   Then ga_true
   Else (copy >>> (first copy >>> (swap >>>
        ga_true [‘x1; copy >>> (first copy >>>
        (swap >>> (drop >>> id) [(first ((first
        (‘x0) >>> second ((first ga_true >>>
        second id) >>> id)) >>> ga_true) >>>
        second ((first (‘x1) >>> second id) >>>
        id)) >>> (‘x); drop >>> id]))])))
   in (‘x)
```

Figure 6. The `pow` function's abstract syntax tree and the result of running the `translate` procedure corresponding to the rightmost column of Figure 5 on it. Note that the resulting abstract syntax tree does not contain any brackets or escapes; they have all been translated to equivalent `GArrow` operations.

system can determine if a function application is *fully saturated*. This is achieved via the $\mathsf{firstClass}(\tau,\vec{\eta})$ hypothesis in App_0, which prevents any function application from producing a non-first-class value via unsaturated application.

The App_{n+1} rule handles n-ary application for $n{\geq}1$. The first hypothesis is standard; the second ensures that a function is never applied to a non-first-class value; the third is standard and the fourth can be thought of as a recursive appeal to App_n. Note that this rule does not assume that the three subderivations take place under the same context. In fact, they must take place under separate contexts; this will matter if Contr is removed.

The Brak and Esc rules are standard, copied from [**?**]. Briefly, they prevent one piece of code from being spliced into another using the ~e construct unless both pieces of code are of the same depth (number of surrounding brackets minus number of surrounding escapes is the same) and their level names are the same. The latter point will matter once a type is introduced for *closed code* in Section 4.7.

4. The Translation

The translation from multi-stage programs to generalized arrows is given by the rightmost column of Figure 5, and is formalized by the function `translate` in `GArrow.v`. Note that the translation operates on *proofs of well-typedness* rather than expressions.

The accompanying Coq formalization in `GArrow.v` includes an inductive type representing each of the productions in Figure 4, using a PHOAS [**?**] representation for expressions. Also included is an inductive type `HasType` of typing derivations under the rules of Figure 5, and a procedure `translate`, which produces a `GArrow` expression by structural recursion on a `HasType` proof. An abstract

syntax tree for the `pow` function is also included, and a corresponding `HasType` for it. The result of applying the translation procedure to a proof tht the `pow` function is well-typed can be found in Figure 6.

Remark 5 The fact that the translation operates on proofs rather than abstract syntax trees has two curious practical consequences in the accompanying `GArrow.v`. The first is that `HasType` must belong to `Set` rather than `Prop`, because although its inhabitants are proofs their identities are not irrelevant. The second is that the unpleasant work of using the structural rules to re-arrange contexts is easily automated using tacticals and the `Ltac` scripting language[2].

The `GArrow.v` formalization covers all material up to this point; the remaining material is not included in the machine-checked portion of this paper except where explicitly stated otherwise.

The remaining subsections will investigate possible object language features which might be added, and the corresponding translation of each feature into generalized arrows. Each of the following subsections is completely independent of the others; any combination of the rule sets can be unioned with the rule set of Figure 5 to produce an object language with that specific combination of features.

4.1 Recursive Let Bindings in Specific Stages

Figure 7 gives syntax, typing rules, and translation rules for the ability to permit recursion at specific levels and types. Note that the predicate recOk is parameterized over both the level $\vec{\eta}$ and the type τ_x where the recursion occurs. This can be useful for:

- Allowing recursion only at certain stages. For example, only in the metalanguage by adding the rule with no hypotheses and $\mathsf{recOk}(\tau, \cdot)$ as the conclusion.
- Allowing recursion only at certain types. For example, allowing recursively-defined functions but not recursively-defined ground values at level $\vec{\eta}$ by adding the rule with no hypotheses and $\mathsf{recOk}(\tau \to \tau, \vec{\eta})$ as the conclusion.

If recursion is to be used at any stage other than the first, it is necessary for the `GArrow` to also be a `GArrowLoop` and implement the `loop` function of Figure 7. This operation must satisfy the laws shown in Figure 8, adapted from [**?** , Figure 7]. These axioms first arose in work on traces on categories [**?**], and were first applied to functional programming in the context of value-recursive monads [**?**].

4.2 Booleans and Branching

Figure 9 gives grammar, typing rules, and translation rules for boolean values and branching. Note again that the conditional and branches of the `if` construct are typed under disjoint pieces of the combined Γ_i, Γ context rather than under a shared context.

4.3 Cross-Stage Persistence

Figure 10 gives the rules for cross-stage persistence (CSP). CSP is permitted only for fully-normalized values belonging to a non-function (ground) type; these types are distinguished by the $\mathsf{reifiable}(\tau, \vec{\eta})$ judgement. Appropriate inference rules must be added for whatever kinds of types (primitives, products, coproducts, etc) are in the system to ensure that $\mathsf{reifiable}(\tau, \vec{\eta})$ is derivable for those types at which it is appropriate.

[2] This turned out to be far easier than expected

$$e ::= \texttt{let}\ x\texttt{=}e\ \ \texttt{in}\ \ e \mid \ldots$$

$$\Sigma ::= \mathsf{recOk}(\tau, \vec{\eta}) \mid \ldots$$

RULE	SYNTAX	SEMANTICS
Rec	$\dfrac{\mathsf{recOk}(\tau_x, \vec{\eta}) \quad x{:}\tau_x^{\vec{\eta}}, \Gamma_x \vdash e_x : \tau_x^{\vec{\eta}} \quad x{:}\tau_x^{\vec{\eta}}, \Gamma_e \vdash e : \tau^{\vec{\eta}}}{\Gamma_x, \Gamma_e \vdash \texttt{let}\ x\texttt{=}e_x\ \texttt{in}\ e : \tau^{\vec{\eta}}}$	$= \Delta_x$; $= \Delta_e$; $=$ `first` (`loop` (Δ_x `>>>` `copy`)) `>>>` Δ_e

```
Class GArrowLoop ((**):Set->Set->Set)
                 ((~>):Set->Set->Set)
                 (  ga:GArrow (**) (~>)) :=
    loop : (a**c~>b**c) -> (a~>b)
```

Figure 7. Typing Rules for Recursive `let` at Specific Stages. Assumes additional judgements for those stages at which recursive let-bindings are permitted.

$$\texttt{loop}\ (\texttt{first}\ h \ggg f) = h \ggg \texttt{loop}\ f$$

$$\texttt{loop}\ (f \ggg \texttt{first}\ h) = \texttt{loop}\ f \ggg h$$

$$\texttt{loop}\ (\texttt{loop}\ f) = \texttt{loop}\ (\texttt{cossa} \ggg \texttt{f} \ggg \texttt{assoc})$$

$$\texttt{second}\ (\texttt{loop}\ f) = \texttt{loop}\ (\texttt{assoc} \ggg \texttt{second}\ \texttt{f} \ggg \texttt{cossa})$$

Figure 8. Laws for the `loop` function. These follow the laws of [**?** , Figure 7], with “Extension” and “Sliding” omitted.

4.4 Product Types in the Object Language

Figure 11 gives rules for product types.

The laws given are exactly those needed to ensure that the `<*>` operator induces a *finite product* (Definition 7) structure with $!X =$ `drop` and $\Delta_X =$ `delta`. FIXME: should the GArrow itself choose `unit`?

Remark 6 Note that `**` and $\otimes$ are not the same. The `**` operator represents *contexts*, which are not first-class in the object language. The $\otimes$ operator represents products, which *are* first-class in the object language.

`Arrows` do not make the distinction above, which is a source of limitations. For example, an `Arrow` for stream processors does not distinguish between a *pair of streams* and a *stream of pairs*; both are `a*b~>c*d` (which is a retract of `(a~>c)*(b~>d)` in the absence of side effects). With `GArrows` *pairs of streams* have type `a**b~>c**d` and *streams of pairs* have type `a`$\otimes$`b~>c`$\otimes$`d`. In a *synchronous dataflow* environment these two concepts coincide; this explains why all existing literature on using `Arrows` for stream processing [**? ?**] and digital circuits [**? ?**] applies only to synchronous environments. Attempts to create `Arrows` for unrestricted Petri Nets

$$\tau ::= \texttt{bool} \mid \ldots$$
$$e ::= \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \ldots$$

RULE	SYNTAX	SEMANTICS
Bool	$\dfrac{}{\mathsf{firstClass}(\texttt{bool}, \vec{\eta})}$	
True	$\dfrac{}{\top \vdash \texttt{true} : \texttt{bool}^{\vec{\eta}}}$	
False	$\dfrac{}{\top \vdash \texttt{false} : \texttt{bool}^{\vec{\eta}}}$	
If	$\dfrac{\Gamma_i \vdash e_i : \texttt{bool}^{\vec{\eta}} \quad \Gamma \vdash e_t : \tau^{\vec{\eta}} \quad \Gamma \vdash e_e : \tau^{\vec{\eta}}}{\Gamma_i, \Gamma \vdash \texttt{if}\ e_i\ \texttt{then}\ e_t\ \texttt{else}\ e_e : \tau^{\vec{\eta}}}$	$= \Delta_i$; $= \Delta_t$; $= \Delta_e$; $= (\texttt{first}\ \Delta_i)$ >>> $(\texttt{branch}\ \Delta_t\ \Delta_e)$

```
Class GArrowBool ((**):Set->Set->Set)
                 ((~>):Set->Set->Set)
                 (  ga:GArrow (**) (~>)) :=
  branch : (a~>b) -> (a~>b) -> ((bool**a)~>b)
```

Figure 9. Typing Rules for booleans.

$$e ::= \%e \mid \ldots$$
$$\Sigma ::= \mathsf{reifiable}(\tau, (\eta, \vec{\eta})) \mid \ldots$$

RULE	SYNTAX	SEMANTICS
CSP	$\dfrac{\mathsf{reifiable}(\tau, \vec{\eta}) \quad \Gamma \vdash e : \tau^{\vec{\eta}}}{\Gamma \vdash \%e : \tau^{\eta, \vec{\eta}}}$	$= \texttt{reify}\ e$

```
Class GArrowReify ((**):Set->Set->Set)
                  ((~>):Set->Set->Set)
                  (  ga:GArrow (**) (~>)) :=
  reify : (a->b) -> (a~>b)
  reify_extensional :
   forall {a}{b}{f:a->b}{g},
    (forall x, (f x)=(g x))
    -> (reify f)~~(reify g)
```

Figure 10. Typing rules for cross-stage persistence (CSP).

[?] are impeded by this limitation. The need to have distinct types for "stream of pairs" and "pair of streams" led the Fudgets library to co-opt the *coproduct* structure of the underlying type system to represent pairs of streams, which explains the anomoly that Paterson notes [? , Section 5.1] in the type of the Fudgets `loop` function [?].

4.5 Coproduct Types in the Object Language

Figure 12 gives the rules for coproduct types. The `branch` and `bool` of Section 4.2 can be seen as a restricted form of `c_merge` and `<+>`.

4.6 Affine, Linear, and Ordered Types in the Object Language

Affine types in the object language can be modeled by omitting `copy` (eliminating the Cont rule); linear types can be simulated by omitting `copy` and `drop` (eliminating the Weak rule). Ordered linear types [?] can be imitated by omitting `swap` (eliminating the Exch rule).

Remark 7 If `swap` is omitted, the definition of `cossa` is no longer redundant, and it must be defined separately.

Typechecking and type inference for affine, linear, and ordered types is a complex topic. This paper does not attempt to address these questions; it takes the finished typing derivation as a starting point for the translation procedure.

4.7 The `eval` Primitive

The rules for `eval` (also called `run`) can be found in Figure 13. The `eval` primitive can only be used safely on *closed code*; the `open` and `close` primitives are needed to mark such regions [?].

The `GArrowEval` class, which has a `Prop` index but no methods, has a close relationship to Haskell's `runST`, the *strict state monad* [?] which has rank-2 type:

```
runST :: (forall s. ST s a) -> a
```

The `runST` function has this type in order to ensure that values returned by `runST` do not contain "dangling references" to the state index `s`. This effect is achieved by taking advantage of the fact that the introduction rule for $e : (\forall\alpha)\tau$ requires that α not appear in the type environment – it is a closedness condition, albeit upon types rather than values (no matter: parametricity supplies the linkage). This closedness condition on types and values closely paralells the closedness conditions in the hypothesis of the Close rule, which must be applied before `eval`.

Theorem 2 The translation converts staged values of *closed* type $\langle\!\langle \tau^{\square} \rangle\!\rangle$ to expressions of a rank-2 type parametric over the `GArrow` instance.

Proof. in `translation_of_closed_code_is_parametric` in `GArrow.v` □

5. Examples

5.1 Exponentiation of Natural Numbers

It is now time to return to the example program, `pow`, expressed using staging annotations:

$\tau ::= \tau \otimes \tau \mid \ldots$

$e ::= \texttt{fst}\ e \mid \texttt{snd}\ e \mid \langle e, e\rangle \mid \ldots$

RULE	SYNTAX	SEMANTICS
$\mathsf{FC}_{\mathsf{prod}}$	$\dfrac{\mathsf{firstClass}(\tau_1,\vec{\eta}) \quad \mathsf{firstClass}(\tau_2,\vec{\eta})}{\mathsf{firstClass}(\tau_1 \otimes \tau_2,\vec{\eta})}$	
Fst	$\dfrac{\Gamma \vdash e : (\tau_1 \otimes \tau_2)^{\vec{\eta}}}{\Gamma \vdash \texttt{fst}\ e : \tau_1^{\vec{\eta}}}$	Δ $=$ `lift(id**drop) >>> iso1 >>>` Δ
Snd	$\dfrac{\Gamma \vdash e : (\tau_1 \otimes \tau_2)^{\vec{\eta}}}{\Gamma \vdash \texttt{snd}\ e : \tau_2^{\vec{\eta}}}$	Δ $=$ `lift(drop**id) >>> iso2 >>>` Δ
Prod	$\dfrac{\Gamma_1 \vdash e_1 : \tau_1^{\vec{\eta}} \quad \Gamma_2 \vdash e_2 : \tau_2^{\vec{\eta}}}{\Gamma_1,\Gamma_2 \vdash \langle e_1, e_2\rangle : (\tau_1 \otimes \tau_2)^{\vec{\eta}}}$	Δ_1 Δ_2 $=$ `lift ( first` Δ_1 `>>> second` Δ_2`)`

```
Class GArrowProd (g:GArrow G)
                  ((<*>):Set->Set->Set) :=
  unit  : Set

  delta : a ~> a<*>a
  iso1  : a<*>unit ~> a
  iso2  : unit<*>a ~> a
  lift  : (a**b)~>(c**d) -> (a<*>b)~>(c<*>d)
  id ~~ delta >>> (lift (id   *** drop)) >>> iso1
  id ~~ delta >>> (lift (drop *** id  )) >>> iso2
```

Figure 11. Product Types

$\tau ::= \tau \oplus \tau \mid \ldots$

$e ::= \texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{case}\ e\ \texttt{of}\ \texttt{|}\ \texttt{L}\ x\ \texttt{->}\ e\ \texttt{|}\ \texttt{R}\ x\ \texttt{->}\ e \mid \ldots$

RULE	SYNTAX	SEMANTICS
$\mathsf{FC}_{\mathsf{coprod}}$	$\dfrac{\mathsf{firstClass}(\tau_1,\vec{\eta}) \quad \mathsf{firstClass}(\tau_2,\vec{\eta})}{\mathsf{firstClass}(\tau_1 \oplus \tau_2,\vec{\eta})}$	
InL	$\dfrac{\Gamma \vdash e : \tau_1^{\vec{\eta}}}{\Gamma \vdash \texttt{inl}\ e : (\tau_1 \oplus \tau_2)^{\vec{\eta}}}$	$= \Delta$ $=$ `iso1 >>> lift(id**codrop) >>>` Δ
InR	$\dfrac{\Gamma \vdash e : \tau_2^{\vec{\eta}}}{\Gamma \vdash \texttt{inr}\ e : (\tau_1 \oplus \tau_2)^{\vec{\eta}}}$	$= \Delta$ $=$ `iso2 >>> lift(codrop**id) >>>` Δ
CP	$\dfrac{\Gamma_0 \vdash e_0 : (\tau_1 \oplus \tau_2)^{\vec{\eta}} \quad \Gamma, x{:}\tau_1^{\vec{\eta}} \vdash e_1 : \tau^{\vec{\eta}} \quad \Gamma, x{:}\tau_2^{\vec{\eta}} \vdash e_2 : \tau^{\vec{\eta}}}{\Gamma_0, \Gamma \vdash \texttt{case}\ e_0\ \texttt{of}\ \texttt{|}\ \texttt{L}\ x\texttt{->}\ e_1\ \texttt{|}\ \texttt{R}\ x\texttt{->}\ e_2 : \tau^{\vec{\eta}}}$	$= \Delta_0$ $= \Delta_1$ $= \Delta_2$ $=$ `lift ( first` Δ_1 `>>> second` Δ_2`) >>> codelta`

```
Class GArrowCoprod (g:GArrow G)
                  ((<+>):Set->Set->Set) :=
  void    : Set           (* the uninhabited type *)
  codrop  : void ~> a
  codelta : a<+>a ~> a
  iso1    : a ~> a<+>void
  iso2    : a ~> void<+>a
  lift    : (a**b)~>(c**d) -> (a<+>b)~>(c<+>d)
  id ~~ iso1 >>> (lift (id    *** codrop)) >>> codelta
  id ~~ iso2 >>> (lift (codrop *** id  )) >>> codelta
```

Figure 12. Coproduct Types

```
pow n x =
  if n==0
  then <[ 1 ]>
  else <[ ~x * ~(pow (n-1) x) ]>
```

Theorem 3 For any $\vec{\eta}$, there exists a typing derivation using the rules of Figures 5 and 9 for $\Gamma \vdash$ `pow : Int->`$\langle\!\langle$`Int`$\rangle\!\rangle$`->`$\langle\!\langle$`Int`$\rangle\!\rangle^{\vec{\eta}}$ where Γ contains suitable type assumptions for `0`, `1`, `(*)`, `(-)`, and `(==)`.

Proof. in pow_hastype in `GArrow.v` □

5.2 BiArrows

`BiArrow`s are meant to model `Arrow`s with a notion of *inversion*. They were introduced in [?] and further examined in [?]. Briefly,

```
Class BiArrow ((~>):Set->Set->Set)
              (arrow:Arrow (~>)) :=

  biarr : (a->b) -> (b->a) -> (a~>b)
  inv   :    a~>b -> b~>a

  pf0   :    inv (biarr f f') ~~ biarr f' f
  pf1   :         inv (inv f) ~~ f
  pf2   :       inv (g >>> f) ~~ (inv f) >>> (inv g)
  pf3   :         inv (arr f) ~~ (arr swap)
  pf4   :       inv (first f) ~~ first (inv f)
```

The `BiArrow` class adds a new constructor `biarr`, which is to be used in place of `arr`. It takes a pair of functions which are required to be mutual inverses. The `inv` function attempts to invert a `BiArrow`.

Types belonging the class `BiArrow` consist of operations which *might be* invertible. Some `BiArrow` values are actually not invert-

$\tau ::= \langle\!\langle \tau^{\Box} \rangle\!\rangle \mid \ldots$

$e ::= \texttt{open}\ e \mid \texttt{close}\ e \mid \texttt{eval}\ e \mid \ldots$

RULE	SYNTAX	SEMANTICS
Open	$\dfrac{\Gamma \vdash e : \langle\!\langle \tau^{\Box} \rangle\!\rangle^{\vec{\eta}}}{\Gamma \vdash \texttt{open}\ e : \langle\!\langle \tau^{\eta'} \rangle\!\rangle^{\vec{\eta}}}$	$= \Delta$ $= \Delta$
Close	$\dfrac{\eta' \notin \mathsf{FV}(\Gamma, \vec{\eta}, \tau) \quad \Gamma \vdash e : \langle\!\langle \tau^{\eta'} \rangle\!\rangle^{\vec{\eta}}}{\Gamma \vdash \texttt{close}\ e : \langle\!\langle \tau^{\Box} \rangle\!\rangle^{\vec{\eta}}}$	$= \Delta$ $= \Delta$
Eval	$\dfrac{\Gamma \vdash e : \langle\!\langle \tau^{\Box} \rangle\!\rangle^{\vec{\eta}}}{\Gamma \vdash \texttt{eval}\ e : \tau^{\vec{\eta}}}$	$= \Delta$ $= \texttt{eval}\ \Delta$

```
Class GArrowEval ((**):Set->Set->Set)
                 ((~>):Set->Set->Set)
                 (  ga:GArrow (**) (~>)) :=
                 (idx:Prop) := { }.
eval : forall ((**):Set->Set->Set)
              ((~>):Set->Set->Set)
              (  ga:GArrow (**) (~>)),
    (forall (idx:Prop),
          (GArrowEval (**) (~>) ga idx) -> (a~>b))
    -> (a->b).
```

Figure 13. Rules for `eval`.

ible, so the `inv` operation is only partial and may fail at runtime. The type system is not capable of ensuring that "well-typed programs cannot go wrong" in this way. Unfortunately there is no way to fix this within the framework of `Arrow`s, because the `Arrow` type class requires that `arr` be defined for arbitrary functions – even those like `fst` (the first projection of a tuple) which cannot possibly have an inverse. Moreover, the `arr` function is tightly woven in to the laws which prescribe the behavior of `Arrow`s, so solving the problem is not as simple as replacing `arr` with `biarr`.

However, one *can* create a `GArrow` which preserves invertibility. There are two possibilities, in fact:

- Realize the `GArrow drop` method using the *logging translation* of [**?** , Section 6], which implements tuple projection by concealing the non-projected coordinates rather than discarding them entirely.
- Declare a superclass of `GArrow` which omits the `drop` function. This is not nearly as violent a change as attempting to remove `arr` from `Arrow`; the translation of Figure 5 remains intact for any derivation which does not use the Weak rule. As a result, object programs typeable under certain variants of linear logic remain translatable.

5.3 Circuit Description

Many researchers have investigated the use of functional programming languages to describe hardware circuits [**? ? ? ? ?**]. The allure is strong: combinational circuits and pure functions have much in common. However, in order to create usable circuits one must allow for sharing and feedback, and this is where the similarities end.

Pure functional languages which represent circuit nodes as first-class language values must add an impurity, *observable sharing* [**?**], to the language in order to preserve sharing information and permit introspection on circuits with feedback. This impurity is incompatible with optimizations present in many compilers for pure functional languages and considerably complicates the semantics of the language. The alternative is to represent circuits using a value-recursive monad [**?**] or `Arrow`; this avoids the pitfalls of observable sharing but requires that circuits be constructed in an object language which is completely different from the functional metalanguage – a choice which dilutes the benefits sought.

With the translation from staging annotations to `GArrow`s, programmers can write circuits *and circuit generators* with a single set of binding, scoping, abstraction, and application mechanisms.

6. Categorical Perspective

The time has come to make good on the promise of the paper's subtitle. Technically what will be exhibited in this section is an *equivalence* of categories, but – like every equivalence – this will give an isomorphism of skeletons.

In addition to abstract theorems involving categories, most subsections of this section will include an example involving a category $\mathbb{O}$ whose objects are the types of some object programming language (pick your favorite side-effect free language) and whose morphisms are the functions of that language.

Definition 1 ([**?** , Definition 2.7]) An object 1 of a category $\mathbb{C}$ is the *terminal object* if there is exactly one morphism into 1 from every other object. This morphism will be written $!A : A \to 1$.

Definition 2 ([**?** , 3.2]) A *binoidal category* is a category $\mathbb{C}$ given with a pair of bifunctors $-\ltimes- : \mathbb{C}\times\mathbb{C} \to \mathbb{C}$ and $-\rtimes- : \mathbb{C}\times\mathbb{C} \to \mathbb{C}$ such that for all objects A, B of $\mathbb{C}$ it is the case that $A \ltimes B = A \rtimes B$, which is also written $A \otimes B$.

Definition 3 ([**?** , 3.3]) A morphism f for which it is the case that $f \ltimes g = f \rtimes g$ for all g is called a *central* morphism.

Binoidal categories are generally used to model computations in which *evaluation order* is significant. The fact that the two bifunctors agree on objects reflects the fact that type systems do not track which coordinate of a tuple was computed first. The fact that the bifunctors may disagree on morphisms reflects the fact that evaluating the left coordinate first may yield a different result than evaluating the right coordinate first. Central maps model computations which are *pure* and therefore commute (in time) with all others. Note that for morphisms f and g the expression $f \otimes g$ is not well-defined unless at least one of f or g is central.

Definition 4 ([**?** , 3.5]) A *premonoidal category* is a binoidal category with an object I such that $A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$ and $X \otimes I \cong X \cong I \otimes X$ for all objects X subject to the coherence conditions of [**?** , p162]. A *strict premonoidal category* is a premonoidal category in which the above isomorphisms are identity maps. A *premonoidal functor* is a functor between premonoidal categories which preserves this structure.

Definition 5 A *symmetric premonoidal category* is a category in which $A \otimes B \cong B \otimes A$ and the mediating isomorphism is its own inverse.

Definition 6 A *monoidal category* is a premonoidal category in which every map is central.

Note that a category may be monoidal in more than one way: there may be multiple bifunctors that satisfy the properties above. For example **Sets**, the category of sets and functions, is monoidal under not only cartesian product but disjoint union as well. The same applies to binoidality and premonoidality.

Definition 7 A *finite product category* is a monoidal category in which $I = 1$ is a terminal object along with a morphism $\Delta_X : X{\to}X{\otimes}X$ for each object X such that the following diagram commutes:

$$\begin{tikzcd}
1{\otimes}X \arrow[d, "\cong"'] & X{\otimes}X \arrow[l, "!X \otimes \mathsf{id}_X"'] \arrow[d, "\mathsf{id}_X \otimes !X"] \\
X \arrow[ur, "\Delta_X"] & 1{\otimes}X \arrow[l, "\cong"]
\end{tikzcd}$$

FIXME: and is equal to the identity – need another branch A *finite product functor* is a functor between finite product categories which preserves this structure.

In a finite product category the monoidal functor will be written $\times$ rather than $\otimes$ to emphasize this additional structure. Note that 1 is the 0-ary product; zero is considered finite in this paper.

Definition 8 ([**?** , Definition B1.2.1(a)]) For $\mathbb{C}$ a category, a $\mathbb{C}$-*indexed category* $\mathbb{D}^{(-)}$ assigns a category $\mathbb{D}^A$ to each object A of $\mathbb{C}$ and a functor $\mathbb{D}^f : \mathbb{D}^X \to \mathbb{D}^Y$ to each morphism $f : X \to Y$ of $\mathbb{C}$ in such a way that $\mathbb{D}^f \circ \mathbb{D}^g \cong \mathbb{D}^{g \circ f}$. If $\mathbb{C}$ has a terminal object 1, then $\mathbb{C} \cong \mathbb{D}^1$.

Definition 9 ([**?** , Definition B1.2.1(b)]) An $\mathbb{C}$-*indexed functor* $F^{(-)} : \mathbb{D}^{(-)} \to \mathbb{E}^{(-)}$ assigns to each object A of $\mathbb{C}$ a functor $F^A : \mathbb{D}^A \to \mathbb{E}^A$ and to each morphism $f : X \to Y$ a natural isomorphism $F^f : (F^Y \circ \mathbb{D}^f) \cong (\mathbb{E}^f \circ F^X)$ allowing the following diagram to commute up to isomorphism of functors:

$$\begin{tikzcd}
\mathbb{D}^Y \arrow[r, "F^Y"] & \mathbb{E}^Y \\
\mathbb{D}^X \arrow[u, "\mathbb{D}^f"] \arrow[r, "F^X"'] & \mathbb{E}^X \arrow[u, "\mathbb{E}^f"']
\end{tikzcd}$$

Definition 10 For a category $\mathbb{C}$ with monoidal bifunctor $(-){\otimes}(-)$, a $\otimes$-*exponential* is a bifunctor $(-){\Rightarrow}(-)$ such that for each object B of $\mathbb{C}$, the functor $B{\Rightarrow}(-)$ is right adjoint to the functor $(-){\otimes}B$.

An $\otimes$-exponential induces the following isomorphism of Hom-sets:

$$\frac{A{\otimes}B \to C}{\overline{A \to B \Rightarrow C}}$$

Definition 11 A *cartesian closed category* is a finite product category with a $\times$-exponential.

Remark 8 The definition of *exponential* is usually stated in a form specific to cartesian products. The more general definition above will allow investigation of exponentials over monoidal structure which is not necessarily a cartesian product.

6.1 Polynomial Categories

Most algebraists are familiar with the construction whereby one passes from a ring R to the ring $R[x]$ of polynomials with one indeterminate and coefficients from R. A similar construction is possible with categories.

Definition 12 (Provisional) Given a category $\mathbb{C}$ with a terminal object 1, and some object B of $\mathbb{C}$, let the *polynomial category over* $\mathbb{C}$ *in* B, written $\mathbb{C}[x{:}B]$, be the free category obtained by adjoining to $\mathbb{C}$ a new morphism $x : 1{\to}B$ and closing under composition and products of morphisms. The morphisms of $\mathbb{C}[x{:}B]$ are called *polynomials over* $\mathbb{C}$ *in* B. [**?** , Definition 2.5]

Like the free group on a set, this "free category obtained by adjoining a new morphism" can be understood intuitively as the category including $x{:}1{\to}B$ while introducing as few new morphisms and satisfying as few new identities as possible. Terms with free variables in them are best understood as morphisms in a polynomial category, and variable-binding operators as functors from the polynomial category back into the host category. This gives some semantic weight to the notion of a "term definable in terms of some hypothetical of type B" – these are exactly the morphisms of $\mathbb{C}[x{:}B]$.

This paper will generally represent polynomial morphisms (except for the indeterminate x) using lower-case letters with a superscript, such as f^B, as a reminder that f^B belongs to $\mathbb{C}[x{:}B]$ rather than $\mathbb{C}$.

Definition 13 (Provisional) The *weakening functor* of a category $\mathbb{C}$ assigns to each object B of $\mathbb{C}$ a functor $\mathbb{C}^{!B} : \mathbb{C}{\to}\mathbb{C}[x{:}B]$ from $\mathbb{C}$ to the polynomial over $\mathbb{C}$ in B such that $\mathbb{C}^{!B}$ is the inclusion functor when $\mathbb{C}$ is regarded as a subcategory of $\mathbb{C}[x{:}B]$.

Remark 9 If it happens that $\mathbb{C}$ is a finite product category, one can construct $\mathbb{C}[x{:}B]$ and the weakening functor explicitly: the weakening functor sends each object A to $B{\times}A$ and each morphism f to $\mathsf{Id}_B{\times}f$. $\mathbb{C}[x{:}B]$ is the subcategory of $\mathbb{C}$ which is the range of this functor. However, if $\mathbb{C}$ has a weaker monoidal structure (perhaps only premonoidal), or none at all, the notion of polynomial category is not definable in this manner.

A slightly more rigorous formulation, adapted from [**?** , Remark 2.6], can be given in terms of indexed categories and universal properties:

Definition 14 (Official) For $\mathbb{C}$ a category with a terminal object 1, a *polynomial category* $\mathbb{C}[x{:}-]$ is a $\mathbb{C}$-indexed category such that for every object B, functor $G{:}\mathbb{C}{\to}\mathbb{D}$ and $d{:}1{\to}G(B)$ there exists a unique functor $[x{:=}d]^G(-){:}\mathbb{C}[x{:}B]{\to}\mathbb{D}$ such that $[x{:=}d]^G(x) = d$ and $[x{:=}d]^G{\circ}\mathbb{C}^{!B} = G$.

$$\begin{tikzcd}
\mathbb{C}[x{:}B] \arrow[dr, "\exists[x{:=}d]^G(-)"] & \\
\mathbb{C} \arrow[u, "\mathbb{C}^{!B}"] \arrow[r, "\forall G"'] & \mathbb{D}
\end{tikzcd}$$

The functor $\mathbb{C}^{!B}$ is called the *weakening functor* at B.

Intuitively, this definition says that for a functor sending $\mathbb{C}$ to $\mathbb{D}$ one can choose any morphism d with codomain in the range of G and factor the weakening functor $\mathbb{C}^{!B}$ through the given functor in such a way that x is sent to d.

Example. Recall that each object of $\mathbb{O}$ represents a type in the object programming language. If we pick some type T, then $\mathbb{O}[x{:}T]$ will be a new category, with an object for every type of $\mathbb{O}$. The objects of this new category represent expressions in our object language having a free variable x of type T. So, for example, if `Int` is a type, then $\mathbb{O}[x{:}\texttt{Int}]$ will be the category of expressions with a free variable x of type `Int`, and if `String` is another type,

$$\frac{b : 1 \to B}{\mathsf{lift}_A(b) : A \to A \otimes B} \qquad \frac{f^B : A \to C}{\kappa x{:}B.f^B : A \otimes B \to C}$$

$$(\kappa x{:}B.f^B) \circ \mathsf{lift}_A(b) = [x{:=}b]^G(f)$$

Figure 14. Rules of the κ-calculus, from [**?**]

there will be an object $\mathbb{O}^{!\mathtt{Int}}(\mathtt{String})$ corresponding to `String` in $\mathbb{O}[x{:}\mathtt{Int}]$ representing object language expressions having overall type `String` and a free variable x of type `Int`.

If we pick some function f in our object language, where f is a function that takes an `Int` and returns a `String`, there will be some $f : \mathtt{Int} \to \mathtt{String}$ in $\mathbb{O}$. Now recall that polynomial categories are just a particular kind of indexed category, and indexed categories must assign a functor to each morphism (Definition 9). The polynomial category assigns f a functor $\mathbb{O}^f : \mathbb{O}[x{:}\mathtt{String}] \to \mathbb{O}[x{:}\mathtt{Int}]$. Note that the order of the argument and return type has changed! This functor takes a term with a free variable x of type `String` and yields a term with a free variable x of type `Int`. How does it do this? *By substituting $f(x)$ for x.*

6.2 Contextual Completeness

Definition 15 ([**?**]) A polynomial category is said to be *contextually complete* if its weakening functors each have a left adjoint.

The left adjoint functor will be written $(-)\otimes B \dashv \mathbb{C}^{!B}$. The unit of the adjunction $\eta_{-\otimes B} : (-) \to (-)\otimes B$ has the property that for every $f^B{:}A \to C$ in $\mathbb{C}[x{:}B]$ there exists a $\hat{f}{:}A\otimes B \to C$ in $\mathbb{C}$ such that $f^B = \mathbb{C}^{!B}(\hat{f}) \circ \eta_{A\otimes B}$. Writing $\lambda x{:}B.f^B$ for $\hat{f}$ gives:

$$f^B = \mathbb{C}^{!B}(\lambda x{:}B.f^B) \circ \eta_{A\otimes B}$$

Remark 10 In [**?**], an explicit definition of λf^B is given for any contextually complete category *which also has finite products*; the definition assumes the monoidal structure of $\mathbb{C}$ has projection and morphism-tupling. The construction bears much similarity to typed combinator conversion, but – as that author notes – is completely first-order (in contrast to Curry's [**?**] combinator conversion) and avoids introducing divergent terms (in contrast to Schöenfinkels [**?**]).

Now, select some morphism $b{:}1 \to B$ and generate the functor $[x{:=}b]^{\mathsf{Id}}(-)$ by Definition 14 corresponding to the identity functor on $\mathbb{C}$. It has the following property:

$$\begin{aligned}
f^B &= \mathbb{C}^{!B}(\lambda x f : B.f^B) \circ \eta_{A\otimes B} \\
[x{:=}b]^{\mathsf{Id}}(f^B) &= [x{:=}b]^{\mathsf{Id}}(\mathbb{C}^{!B}(\lambda x{:}B.f^B) \circ \eta_{A\otimes B}) \\
[x{:=}b]^{\mathsf{Id}}(f^B) &= [x{:=}b]^{\mathsf{Id}}(\mathbb{C}^{!B}(\lambda x{:}B.f^B)) \circ [x{:=}b]^{\mathsf{Id}}(\eta_{A\otimes B}) \\
[x{:=}b]^{\mathsf{Id}}(f^B) &= (([x{:=}b]^{\mathsf{Id}} \circ \mathbb{C}^{!B})(\lambda x{:}B.f^B)) \circ [x{:=}b]^{\mathsf{Id}}(\eta_{A\otimes B}) \\
[x{:=}b]^{\mathsf{Id}}(f^B) &= \mathsf{Id}_{\mathbb{C}}(\lambda x{:}B.f^B) \circ [x{:=}b]^{\mathsf{Id}}(\eta_{A\otimes B}) \\
[x{:=}b]^{\mathsf{Id}}(f^B) &= (\lambda x{:}B.f^B) \circ [x{:=}b]^{\mathsf{Id}}(\eta_{A\otimes B})
\end{aligned}$$

The last two steps exploit the universal property $[x{:=}b]^{\mathsf{Id}} \circ \mathbb{C}^{!B} = \mathsf{Id}_{\mathbb{C}}$ of the weakening functor (Definition 14).

Following [**?**], abbreviate $\mathsf{lift}_A(b) \stackrel{\text{def}}{\equiv} [x{:=}b]^{\mathsf{Id}}(\eta_{A\otimes B})$. The above definitions and derivations give the three rules of the κ-calculus introduced in [**?**] to isolate the "first order" element of the lambda calculus. These rules are shown in Figure 14.

These inference rules define the syntax of the κ-calculus, and the derivation shows that any syntactical term of the calculus identifies a morphism in a contextually complete category. The κ-calculus is a syntax for the internal language of a contextually complete category in the same way that λ-calculus is a syntax for the internal language of a cartesian closed category.

6.3 Reification

Having reviewed polynomial categories and the standard definition of contextual completeness, how can one reason about programs which *manipulate* other programs with free variables? Answer: *reification* of categories.

Just as polynomial categories were a particular kind of indexed category, reification of one category in another is a particular kind of *indexed functor* between their polynomial categories.

Definition 16 If $\mathbb{O}[x{:}-]$ and $\mathbb{M}[x{:}-]$ are polynomial categories and $\llparenthesis\cdot\rrparenthesis{:}\mathbb{O} \to \mathbb{M}$ is a functor, $\mathbb{M}$ *reifies* $\mathbb{O}$ via $\llparenthesis\cdot\rrparenthesis$ if there is an indexed functor

$$\llparenthesis\cdot\rrparenthesis^{(-)} : \mathbb{O}[x{:}-] \to \mathbb{M}[x{:}\llparenthesis - \rrparenthesis]$$

such that for each object B of $\mathbb{O}$ the following diagram commutes up to isomorphism of functors:

$$\begin{tikzcd}
\mathbb{O}[x{:}B] \arrow[r, "\llparenthesis\cdot\rrparenthesis^B"] & \mathbb{M}[x{:}\llparenthesis B \rrparenthesis] \\
\mathbb{O} \arrow[u, "\mathbb{O}^{!B}"] \arrow[r, "\llparenthesis\cdot\rrparenthesis"'] & \mathbb{M} \arrow[u, "\mathbb{M}^{!\{B\}}"']
\end{tikzcd}$$

Remark 11 Two technicalities must be noted, but can be skipped on a first reading. First, the above abuses notation somewhat: $\llparenthesis\cdot\rrparenthesis$ is not strictly the same thing as $\llparenthesis\cdot\rrparenthesis^{(-)}$; the former is a non-indexed functor, the latter an $\mathbb{O}$-indexed functor. The notation is recycled because the two have similar effect. Second, $\mathbb{M}[x{:}-]$ is not the same thing as $\mathbb{M}[x{:}\llparenthesis - \rrparenthesis]$; the latter is the indexed category resulting from *reindexing* the former along the functor $\llparenthesis\cdot\rrparenthesis$. Similar notation was chosen in order to de-emphasize the least important details.

Example. Let $\mathbb{M}$ be a category whose objects are the types of the metalanguage and whose morphisms are its functions; this means that $\mathbb{M}[x{:}-]$ has an object for every type of the *metalanguage*. The functor $\llparenthesis\cdot\rrparenthesis : \mathbb{O} \to \mathbb{M}$ must assign a *metalanguage* type to each *object language* type, so in a certain sense the metalanguage has a copy of the object language type system within it. Reindexing the polynomial category $\mathbb{M}[x{:}-]$ by $\llparenthesis\cdot\rrparenthesis$ to form $\mathbb{M}[x{:}\llparenthesis - \rrparenthesis]$ essentially means focusing attention on the subset of our metalanguage whose free variable types and return types are all drawn from this copy of the object language's types. Now, consider the properties bestowed by the indexed functor. For any object $B \in \mathbb{O}$, the component of the indexed functor will give a non-indexed functor

$$\llparenthesis - \rrparenthesis^B : \mathbb{O}[x{:}-] \to \mathbb{M}[x{:}\llparenthesis - \rrparenthesis]$$

What does this functor do? The last part of Definition 16 requires that the functor supplied for each object has essentially the same behavior as the $\llparenthesis\cdot\rrparenthesis$ functor combined with $\mathbb{M}[x{:}-]$'s weakening functor $\mathbb{M}^{!B}$. So if X is an object of $\mathbb{O}$ and $\mathbb{O}^{!B}(X)$ is the result of weakening X into $\mathbb{O}[x{:}B]$, then reifying this give the same thing as weakening $\llparenthesis X \rrparenthesis$ into $\mathbb{M}[x{:}\llparenthesis B \rrparenthesis]$:

$$\llparenthesis \mathbb{O}^{!B}(X) \rrparenthesis^B \cong \mathbb{M}^{!\{B\}}(\llparenthesis X \rrparenthesis)$$

This is why similar notation was chosen for $\llparenthesis\cdot\rrparenthesis$ and $\llparenthesis\cdot\rrparenthesis^{(-)}$. Definition 9 says that for a morphism $f{:}X \to Y$ in $\mathbb{O}$, there will be a functor $\mathbb{O}^f : \mathbb{O}[x{:}Y] \to \mathbb{O}[x{:}X]$. It was determined earlier that this

functor has the effect of substituting $f(x)$ for x in a term that has a free variable x. Moving now to the reification functor, it is clear that $\langle\!| f |\!\rangle^B : \mathbb{M}[x{:}\langle\!| Y |\!\rangle] \to \mathbb{M}[x{:}\langle\!| X |\!\rangle]$. But what does *this* functor do?

Recall that an indexed functor also assigns a natural isomorphism to every morphism. Suppose B is an object in $\mathbb{O}$, and X, Y are objects in $\mathbb{O}[x{:}B]$. Then by Definition 9, our reification functor must assign to each $f : X \to Y$ a natural isomorphism

$$\langle\!| - |\!\rangle^f : (\mathbb{M}^{\langle\!| f |\!\rangle} \circ \langle\!| - |\!\rangle^Y) \cong (\langle\!| - |\!\rangle^X \circ \mathbb{O}^f)$$

This is the key to understanding what $\langle\!| f |\!\rangle^B$ does. In prose, the above isomorphism says that applying $\mathbb{O}^f$ and then reifying is the same as reifying *first* and then applying $\langle\!| f |\!\rangle$. So we know that $\langle\!| f |\!\rangle$ has the effect of substituting *under the brackets*, which is exactly the operation needed in order to manipulate object-language programs.

To sum up, starting from a given functor $\langle\!| \cdot |\!\rangle : \mathbb{O} \to \mathbb{M}$, asking for a family of functors, one $\langle\!| \cdot |\!\rangle^B$ for each $B \in \mathbb{O}$ does not say much: these could all be trivial functors which send every object to a single object and every morphism to its identity. Requiring that this family of functors *forms an indexed functor* is what forces $\langle\!| \cdot |\!\rangle^{(-)}$ to have the "substitution under brackets" behavior. The natural isomorphism required by Definition 9 turns into precisely the condition which characterizes the code-splicing behavior of staging annotations.

6.4 Contemplation

Definition 17 A category $\mathbb{M}$ *contemplates* a category $\mathbb{O}$ if $\mathbb{M}$ reifies $\mathbb{O}$ and $\mathbb{M}$ is contextually complete. A category is *contemplatively complete* if it contemplates itself.

Contemplation is the categorical property which best models multi-stage type systems; Contemplative completeness is the categorical property which best models *homogeneous* multi-stage type systems.

Theorem 4 (Staging and Contemplation) The category whose objects are the types of Figure 5 and whose morphisms are the functions definable in that system forms a contemplatively complete category.

Proof. Establish a category $\mathbb{M}$ with an object for each type of the language and for each object B freely generate the polynomial category over $\mathbb{M}$ in B. The inference rules Lam, App_0 and App_{n+1} define the operations of the κ-calculus and satisfy the laws of Figure 14, so contextual closure is straightforward. The syntactical operation which sends an expression e having free variable x of type B to the expression $\langle\!| e[x{:=}(\sim x)] |\!\rangle$ is an indexed functor (with B being the index) whose action on types sends $\mathbb{M}^{!B}(A)$ to $\mathbb{M}^{!\langle\!| B |\!\rangle}(\langle\!| A |\!\rangle)$. This indexed functor is the reification functor with the required properties. ☐

Definition 18 ([**?**]) For a monoidal category $\mathbb{C}$ and endofunctor $F : \mathbb{C} \to \mathbb{C}$, the endofunctor has *functorial strength* if for every pair of objects A, B of $\mathbb{C}$ there is a morphism satisfying certain coherence conditions:

$$F_{A,B} : F(A) \otimes B \to F(A \otimes B)$$

Definition 19 A contemplatively complete category has *enriched contemplation* if the coordinates of the reification functor all have strength.

Strengths on the reification functor give the ability to perform cross-stage persistence. The morphism $\langle\!| \cdot |\!\rangle_{1,A} : \langle\!| 1 |\!\rangle \otimes A \to \langle\!| 1 \otimes A |\!\rangle = A \to \langle\!| A |\!\rangle$ provides the required transition.

6.5 κ-Categories and Freyd Categories

Definition 20 ([**?** , Definition 11]) A κ-category consists of a finite product category $\mathbb{C}$ and a $\mathbb{C}$-indexed category $H^{(-)}$ such that:

1. For each object A of $\mathbb{C}$, H^A has the same objects as $\mathbb{C}$, and H^f is the identity on objects.
2. For each projection morphism $\pi : B{\times}A{\to}B$ of $\mathbb{C}$, H^π has a left adjoint $(-){\times}A$
3. For each morphism $f : B \to B'$, the natural transformation $\phi : ((-){\otimes}B) \circ H^{f \times \mathsf{id}_A} \to H^f \circ ((-){\otimes}B')$ induced by the adjointness in the previous bullet point is in fact an isomorphism.

Theorem 5 Categories with enriched contemplation and finite products are in bijective correspondence with κ-categories.

Proof. Given a category $\mathbb{M}$ with enriched contemplation and finite products, $\mathbb{M}[x{:}-]$ is the requisite $\mathbb{M}$-indexed category, (1) each $\mathbb{M}[x{:}B]$ has the same objects as $\mathbb{M}$ and the weakening functor $\mathbb{M}^{!B}$ is identity-on-objects (Definition 14), (2) because $\mathbb{M}$ is contemplative it is contextually complete (Definition 17), so the weakening $\mathbb{M}^\pi$ of any projection morphism π has left adjoint (Definition 15), and (3) the natural isomorphism imposed by the indexed reification functor (Definition 16) supplies the requisite ϕ. ☐

Definition 21 ([**?** , A.4]) A *Freyd Category* is a category $\mathbb{C}$ with finite products, a symmetric premonoidal category $\mathbb{K}$, and an identity-on-objects strict symmetric premonoidal functor $J : \mathbb{C} \to \mathbb{K}$.

Theorem 6 ([**?** , Theorems 13 and 14]) Freyd Categories and κ-categories and are in bijective correspondence.

Theorem 7 (The Stages-Arrows Isomorphism) Categories with enriched contemplation and finite products are in bijective correspondence with Freyd categories.

Proof. By transitivity of bijective correspondence. ☐

Remark 12 The proof shown for Theorem 7 is clearly trivial once the appropriate context has been set up. The main contribution of this section is not a one-line proof, but rather the identification and definition of *enriched contemplation* as the appropriate criterion. Specifically, enriched contemplation is a strong enough condition to make the proof of bijective correspondence go through (almost effortlessly), but still weak enough that a large class of stage-annotated metaprogramming languages constitute categories with enriched contemplation. Furthermore, enriched contemplation is not even quite so important as the weaker forms it suggests. If categories with enriched contemplation and finite products are in bijective correspondence with Freyd categories, it is natural to ask what is in bijective correspondence with obvious weakenings such as monoidal categories with enriched contemplation, premonoidal categories with enriched contemplation, categories with non-enriched contemplation, and categories which reify categories besides themselves. Generalized arrows subsume all of these. So while Theorem 7 may not be surprising or unlikely, the connection it establishes justifies the generalization.

7. Future Work

7.1 Polymorphism and Inference

The presentation in this paper did not cover either type polymorphism or inference; these will be necessary for a production-quality

```
id_left    : forall (A B:Set)    (f:A~>B),                                  id >>> f ~~ f
id_right   : forall (A B:Set)    (f:A~>B),                                         f ~~ f >>> id
comp_assoc : forall (A B C D:Set)(f:A~>B)(g:B~>C)(h:C~>D),         (f >>> g) >>> h ~~ f >>> (g >>> h)
first_law  : forall (A B C D:Set)(f:A~>B)(g:B~>C),                 first (f >>> g) ~~ first(c:=D) f >>> first g
law5       : forall (A B C:Set)  (f:A~>B),               first (first f) >>> assoc ~~ assoc(c:=C)(b:=B) >>> first f
law6       : forall (A B C:Set),                                               cossa ~~ swap >>> assoc (b:=B) >>> swap
law7       : forall (A B C:Set)(f:A~>B),                           first f >>> drop ~~ drop (b:=B) >>> f
law8       : forall (A B:Set),                           swap (b:=B)(a:=A) >>> swap ~~ id
law9       : forall (A B:Set),                                        copy >>> swap ~~ copy (a:=A)
law_assoc  : forall (A B C:Set),              assoc (c:=C)(b:=B)(a:=A) >>> cossa ~~ id
law_cossa  : forall (A B C:Set),              cossa (c:=C)(b:=B)(a:=A) >>> assoc ~~ id
```

Figure 15. `GArrow` laws of Figure 3, rendered as Coq propositions to be satisfied by any `Instance` of `GArrow`

system. This will require extending the grammar for types:

$$\alpha ::= \text{ type variables}$$
$$\tau ::= \dots \mid \alpha \mid \forall\alpha.\tau$$

The $\mathsf{firstClass}(\tau,\vec{\eta})$, $\mathsf{reifiable}(\tau,\vec{\eta})$, and $\mathsf{recOk}(\tau,\vec{\eta})$ judgements present a small complication for polymorphism; when attempting to assign a polymorphic type to an expression, the typical rule used [**?**] is something similar to:

$$\frac{\begin{array}{c}\alpha \notin \mathsf{FV}(\Gamma_1,\Gamma_2,\tau_2,\vec{\eta}) \\ \Gamma_1 \vdash e_1 : \tau_1^{\vec{\eta}} \\ \Gamma_2, x : (\forall\alpha.\tau_1)^{\vec{\eta}} \vdash e_2 : \tau_2^{\vec{\eta}}\end{array}}{\Gamma_1,\Gamma_2 \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : \tau_2^{\vec{\eta}}}$$

In this arrangement, the type inference procedure may find itself confronted with the need to prove judgements such as $\mathsf{firstClass}(\alpha,\vec{\eta})$ where α is a type *variable*. The solution to this situation is to introduce qualified types [**?**], gathering a list of constraints imposed on each type variable and annotating type quantifiers with these constraints, creating types such as $\forall\alpha.\mathsf{firstClass}(\alpha,\vec{\eta}) \Rightarrow \tau$.

Level polymorphism will also be necessary for a production-quality system. The algorithm described in [**?**] appears to be the most appropriate. Among the changes required will be extending the grammar for types:

$$\tau ::= \dots \mid \forall\eta.\tau$$

and adding a typing rule to propagate the $\mathsf{firstClass}(\tau,\vec{\eta})$ judgement across level quantifiers:

$$\frac{\begin{array}{c}\eta' \notin \mathsf{FV}(\tau,\vec{\eta}) \\ \mathsf{firstClass}(\tau[\eta\texttt{:=}\eta'],\vec{\eta})\end{array}}{\mathsf{firstClass}(\forall\eta.\tau,\vec{\eta})}\mathrm{FC}_\forall$$

7.2 Dependent Types

The characterization of staging annotations as an indexed functor among polynomial categories gives a category-theoretic foundation to multi-stage programming. In this context, dependent types are understood as the objects of locally cartesian closed categories [**?** , Definition 9.19]. This should provide a straightforward way to investigate multi-stage programming at all corners of the lambda-cube [**?**], perhaps leading to a sound multi-stage Calculus of Constructions [**?**].

References